Slow Sorting: A Whimsical Inquiry

Bryant A. Julstrom
Department of Computer Science
St. Cloud State University
St. Cloud, Minnesota 56301
julstrom@eeyore.stcloud.msus.edu

Abstract

Sorting is one of the most common and important computing operations. In analyzing and comparing sorting algorithms, we consider their execution times, as indicated by the number of operations they execute as they sort n elements. The simplest algorithms have times that grow approximately as n^2, while more complex algorithms offer times that grow approximately as $n \log n$. This paper pursues a contrary goal: a sorting algorithm whose time grows *more quickly* than that of the well-known algorithms. The paper exhibits and analyzes such an algorithm.

1 Introduction

Sorting is one of the most common and important computing operations. It is the focus of a vast literature and the subject of lectures and exercises in courses ranging from the first a computing student takes to the very advanced. Knuth describes and analyzes about 25 sorting algorithms in his classic series [Knuth 1973]. The more familiar sorting algorithms have proper names, like Selection Sort and Heap Sort.

In comparing sorting algorithms, we most often consider the *time* they take, as indicated by the numbers of operations they require to sort n values. We compare the rates of growth of those times as the number of elements to be sorted grows. The simplest algorithms, like Selection Sort, are $O(n^2)$, while more complicated algorithms, like Heap Sort, are generally $O(n \log n)$. We prefer to sort larger sets of elements with the more efficient algorithms, those whose times grow more slowly as n grows.

This paper poses a perverse but entertaining question: Can we write a sorting algorithm which is *less* efficient than those known? That is, is it possible to write a sorting algorithm in which each operation plausibly advances the sorting process but whose time is *greater* than $O(n^2)$?

The following sections describe the context of this puzzle, give the rules for the "slow sorting" game, describe an algorithm which answers the question above in the affirmative, prove that the algorithm's time is $O(n^3)$, and indicate how to build sorting algorithms of arbitrarily large polynomial time complexity.

2 Sorting

As Knuth has observed, *ordering* might be a better name for the process we call *sorting*: rearranging a list of data elements so that the values of the elements (or of a *key* field within each) ascend or descend from the beginning of the list to its end [Knuth 1973].

We divide sorting algorithms into two categories depending on the situations in which they apply. *Internal* sorting orders elements held in the computer's memory, as in an array or a linked list. *External* sorting orders elements held on a mass storage device such as a disk drive. Here we consider the internal sorting of elements held in an array. For simplicity the elements will be single values, say integers; that is, each element is its own key.

We generally place internal sorting algorithms in two categories based on the times they take to sort n elements. The simplest algorithms—Insertion Sort, Selection Sort, Bubble Sort—have two nested loops. The outer loop executes $n - 1$ times while the inner loop carries out an operation that leaves one more element in its proper sorted position relative to the elements already sorted. These algorithms execute a number of operations roughly proportional to n^2; their times are $O(n^2)$.

Sorting algorithms in the second category are more complicated than the n^2 algorithms. We motivate their development by pointing out how quickly the execution times of the n^2 algorithms grow as the number of elements to be sorted grows. The more complicated algorithms have average-case times roughly proportional to $n \log n$; they include Heap Sort, Merge Sort, and Quick Sort (which is still $O(n^2)$ in the worst case).

One algorithm which does not fit into these categories is Shell Sort, which uses Insertion Sort to arrange the elements being sorted into interleaved sorted sequences at intervals of k positions, then decreases k. Shell Sort's average-case time depends on the sequence of interval values and has not been precisely determined. It may be $O(n(\log n)^2)$ or $O(n^{5/4})$ for good choices of intervals [Weiss and Sedgewick 1988].

No general sorting algorithm which compares entire values (or keys) can have average-case time which is less than $O(n \log n)$ [Knuth 1973, Horowitz and Sahni 1978]. On the other hand, is n^2 an upper bound on the time complexity of reasonable sorting algorithms? This paper considers the following quixotic question: can we develop a sorting algorithm which is *slower* than those generally described; that is, whose time complexity is *greater* than $O(n^2)$?

3 The Slow Sorting Puzzle

Our object is to develop an internal sorting algorithm whose time grows more rapidly than $O(n^2)$ with the number of elements n to be sorted. It is trivial to insert into a repeated section of any known algorithm some block of code, whose time depends on n, which doesn't affect the computation. This maneuver violates

the spirit of our inquiry; we define the problem in the following way:

> Algorithms which are candidates for the slow-sorting prize must sort an array A[1..n] of n values by comparing and moving values. Each operation must plausibly advance the sorting process.

4 An Inefficient Sort

The familiar Bubble Sort is generally considered the least efficient of the easy ($O(n^2)$) sorts. In Bubble Sort, each comparison may result in moving two elements of an array a single position closer to their sorted locations. To be less efficient than Bubble Sort and the other n^2 algorithms, a sorting algorithm must move elements towards their final locations in such a way that each step helps those following it very little and the final sorted ordering emerges very slowly. The following algorithm, called Slow Sort, satisfies this requirement.

The Slow Sort algorithm comprises three nested loops. A variable `Interval` controls the outermost loop; `Interval` takes on the values $n/2, n/3, n/4, \ldots, n/n$ (integer division). The inner two loops implement a selection sort of the elements `A[1]`, `A[1+Interval]`, `A[1+2*Interval]`, $\cdots$. (The Appendix gives Modula-2 code for this algorithm.) Recall that Selection Sort scans the elements to identify the smallest value and swap it into the first position, then scans all the elements but the first for the second smallest, which it swaps into the second position, and so on. Here, Selection Sort scans the elements at intervals of `Interval`. Each selection sort step should move the elements of the array that it touches closer to their target (sorted) locations in the array.

For example, consider this array of 10 elements:

	1	2	3	4	5	6	7	8	9	10
A	42	35	66	17	24	31	12	55	28	15

In the first iteration of Slow Sort's outer loop, `Interval` is 5, and the inner two loops selection sort the elements `A[1]` and `A[6]` to obtain the following arrangement of values:

	1	2	3	4	5	6	7	8	9	10
A	31	35	66	17	24	42	12	55	28	15

In the second iteration of the outer loop, `Interval` is 3, and the inner loops sort the elements `A[1]`, `A[4]`, `A[7]`, and `A[10]`:

	1	2	3	4	5	6	7	8	9	10
A	12	35	66	15	24	42	17	55	28	31

This process continues with `Interval` equal to 2 and then 1, after which the array is sorted.

5 Slow Sort's Time

Slow Sort's time is easy to describe. The number of array elements each Selection Sort step sorts is always either the denominator of the expression for `Interval` or that denominator plus one, and within Selection Sort each search for the next smallest value tests all the remaining unsorted elements.

The first traversal of Slow Sort's outer loop, in which `Interval` is $n/2$, Selection Sorts at least two elements. Selection Sort is $O(n^2)$, so the time for this iteration is roughly proportional to 2^2. The next iteration, in which `Interval` is $n/3$, sorts at least three elements (four in the example above) in time proportional to 3^2, and so on. The last iteration, in which `Interval` is 1, selection sorts all the elements, in time proportional to n^2, so the total time for the sorting is roughly proportional to

$$2^2 + 3^2 + 4^2 + \cdots + n^2 = \sum_{i=2}^{n} i^2$$

n	Heap Sort	Selection Sort	Slow Sort
10	44	45	252
20	120	190	2124
30	210	435	7252
40	320	780	17393
50	424	1225	34227

Table 1: The numbers of comparisons executed by Slow Sort, Selection Sort, and Heap Sort on pseudo-random data sets of sizes n = 10, 20, 30, 40, and 50. These results support the claim that the time Slow Sort requires to sort n values is $O(n^3)$.

$$= \frac{n(n+1)(2n+1)}{6} - 1$$
$$= \frac{n^3}{3} + \frac{n^2}{2} + \frac{n}{6} - 1$$

The last expression is dominated by an n^3 term; the time of the algorithm is $O(n^3)$.

Table 1 shows the results of running Selection Sort, Heap Sort, and Slow Sort on pseudo-random value sets of several sizes. Each algorithm reported the number of value comparisons required to sort the values (representing the total number of operations executed). This investigation supports the analysis above: the number of comparisons Slow Sort executes grows as a cubic function of the number of elements being sorted.

Slow Sort thus satisfies the conditions set out in Section 3. Each operation plausibly advances the sorting process, yet the algorithm's time is $O(n^3)$.

6 Variations

The clever reader will have figured out that some other n^2 sorts can replace Selection Sort in Slow Sort to produce an algorithm whose time is $O(n^3)$. Similarly, using an $n \log n$ sort in place of Selection Sort can produce a sort whose time is $O(n^2 \log n)$. In general, algorithms generated in this way can be used as we have used Selection Sort to produce sorting algorithms whose times are $O(n^k)$ or $O(n^k \log n)$ for any $k \geq 2$.

Not every n^2 sort will produce an n^3 algorithm when used in Slow Sort. Consider, for example, the two versions of the familiar Bubble Sort. The simplest version performs every possible "bubble" pass, even after the array elements are sorted, so that the number of comparisons it performs is always the maximum possible. The slightly more efficient version includes a Boolean flag which terminates the algorithm after a bubble pass in which no exchanges occur. When the elements are sorted, the algorithm halts. In the best case (the elements initially sorted), this sort scans the array exactly once.

Using the first version of Bubble Sort to build Slow Sort produces an algorithm whose time is $O(n^3)$, in exactly the same

way as the development above with Selection Sort. Using the second version of Bubble Sort, however, produces a sort which is $O(n^2)$. The Bubble Sort steps move elements towards their final sorted locations, and the Boolean flag terminates each step when there is no more for it to do; this eliminates a vast number of computations, so that the resulting version of Slow Sort is $O(n^2)$.

7 Conclusion

The clever reader also will have asked herself or himself why we should try to find a *less* efficient way to sort, or to perform any operation. If we need an answer, beyond the intellectual entertainment found in this exercise, perhaps it lies here: By aiming intentionally at inefficiency, we may become more adept at identifying and removing poor code from our usual programs, where elegance and speed count.

Slow Sort is slow because it applies the Selection Sort step clumsily, in contrast to the clever application of Insertion Sort in Shell Sort. In Shell Sort, each insertion step within Insertion Sort stops as soon as it has placed its element among those already sorted, and preceeding steps in Shell Sort, with larger increment values, ensure that this happens relatively quickly. Each iteration of Slow Sort's outer loop takes very little advantage of the partial sorting that previous iterations have accomplished. Each step in its Selection Sort always examines every unsorted element among those it must arrange.

The lesson, then, might be this: When we use a segment of code to perform a subtask, we should be careful that following steps take advantage of the subtask's effects rather than fritter them away.

Exercises

1. Implement Selection Sort and show that the number of operations it takes to sort n elements is $O(n^2)$.

2. Consider Slow Sort implemented with Selection Sort as described in Section 4. Create an array `A[1..10]` of 10 values for which, after the first two iterations of Slow Sort's outer loop, the value initially in `A[1]` is **farther** from its target location than it was when the algorithm began. What does this suggest about this version of Slow Sort? Is it necessary that an efficient sorting algorithm always move elements only towards their target locations? (Hint: Consider Heap Sort.)

3. Implement two versions of Slow Sort, one using the simplest version of Bubble Sort within the outermost loop, and the other using the more efficient version of Bubble Sort which includes the Boolean flag. Count the numbers of comparisons each makes on data sets of various sizes. Do these results confirm the discussion at the end Section 6?

4. Another n^2 sort is Insertion Sort, which moves each newly-considered element down into the sorted part of the array, shifting larger elements up as necessary. Implement Slow Sort using Insertion Sort within the outermost loop and count the numbers of comparisons the resulting algorithm makes on random data sets of various sizes. Does this version of Slow Sort appear to be $O(n^3)$? Explain.

5. Rewrite Slow Sort using an $(n \log n)$ sort such as Heap Sort within the outermost loop. Count the numbers of comparisons the resulting algorithm makes as it sorts random data sets of various sizes. What seems to be true of the time of the resulting algorithm?

Appendix

Modula-2 code for the Slow Sort algorithm. `IntArray` is declared to be `ARRAY[1..Max] OF INTEGER`, and `Max` is a program constant. The procedure `Swap` exchanges the values of its two parameters.

```
PROCEDURE SlowSort(VAR A : IntArray;
                       N : INTEGER;
                   VAR C : CARDINAL);
(* An inefficient sort, with time
   which is O(n^3). C returns the
   number of comparisons of array
   elements. *)
  VAR Interval,K,i,j,min : INTEGER;
BEGIN
  C := 0; K := 2; Interval := N DIV K;
  WHILE Interval >= 1 DO
    (* Selection sort *)
    i := 1;
    WHILE i <= N - Interval DO
      min := i;
      j := i + Interval;
      WHILE j <= N DO
        INC (C);
        IF A[j] < A[min] THEN
          min := j
        END;
        j := j + Interval
      END;
      Swap ( A[min],A[i] );
      i := i + Interval;
    END;
    (* End of selection sort *)
    INC (K);
    Interval := N DIV K;
  END (* while *)
END SlowSort;
```

References

Horowitz, Ellis and Sartaj Sahni (1978). *Fundamentals of Computer Algorithms.* Potomac, Maryland: Computer Science Press, Inc.

Knuth, Donald E. (1973). *The Art of Computer Programming, Vol.3: Sorting and Searching.* Reading, Massachusetts: Addison-Wesley Publishing Company.

Weiss, Mark Allen and Robert Sedgewick (1988). Tight Lower Bounds for Shellsort, in *Proceedings of the* 1^{st} *Scandanavian Workshop on Algorithm Design* (R. Karlsson and A. Lingus, Eds.). Berlin: Springer-Verlag.